

深入浅出Flink(2)

一、课前准备

1. 掌握Flink基本特性
2. 掌握Flink编程模型
3. 掌握Flink并行度

二、课堂主题

掌握常见的开发API，为Flink开发打好基础

三、课程目标

1. 掌握常见的DataStream常见的source
2. 掌握常见的DataStream的transformation操作
3. 掌握常见的DataStream的sink操作

四、知识要点(40分钟)

4.1 Flink之数据源 (DataStream)

4.1.1 source简介

source是程序的数据源输入，你可以通过`StreamExecutionEnvironment.addSource(sourceFunction)`来为你的程序添加一个source。

flink提供了大量的已经实现好的source方法，也可以自定义source：

1. 通过实现sourceFunction接口来自定义无并行度的source
2. 通过实现ParallelSourceFunction 接口 or 继承RichParallelSourceFunction 来自定义有并行度的source

大多数情况下，我们使用自带的source即可。

获取source的方式（自带的）

- (1) 基于文件
`readTextFile(path)`
读取文本文件，文件遵循`TextInputFormat` 读取规则，逐行读取并返回。
- (2) 基于socket
`socketTextStream`
从socket中读取数据，元素可以通过一个分隔符切开。
- (3) 基于集合
`fromCollection(Collection)`
通过java 的collection集合创建一个数据流，集合中的所有元素必须是相同类型的。
- (4) 扩展数据源
`addSource` 可以实现读取第三方数据源的数据
系统内置提供了一批connectors，连接器会提供对应的source支持【kafka】

扩展的数据源

- [Apache Kafka](#) (source/sink) 后面重点分析
- [Apache Cassandra](#) (sink)
- [Amazon Kinesis Streams](#) (source/sink)
- [Elasticsearch](#) (sink)
- [Hadoop FileSystem](#) (sink)
- [RabbitMQ](#) (source/sink)
- [Apache NiFi](#) (source/sink)
- [Twitter Streaming API](#) (source)

4.1.2 数据源之collection

```
public class StreamingSourceFromCollection {
    public static void main(String[] args) throws Exception {
        //步骤一：获取环境变量
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //步骤二：模拟数据
        ArrayList<String> data = new ArrayList<String>();
        data.add("hadoop");
        data.add("spark");
        data.add("flink");
        //步骤三：获取数据源
        DataStreamSource<String> dataStream = env.fromCollection(data);
        //步骤四：transformation操作
        SingleOutputStreamOperator<String> addPreStream = dataStream.map(new
MapFunction<String, String>() {
            @Override
            public String map(String word) throws Exception {
                return "kaikeba_" + word;
            }
        });
        //步骤五：对结果进行处理（打印）
        addPreStream.print().setParallelism(1);
        //步骤六：启动程序
        env.execute("StreamingSourceFromCollection");
    }
}
```

4.1.3 自定义单并行度数据源

```
/**
 * 注意：指定数据类型
 * 功能：每秒产生一条数据
 */
public class MyNoParalleSource implements SourceFunction<Long> {
    private long number = 1L;
    private boolean isRunning = true;
    @Override
    public void run(SourceContext<Long> sct) throws Exception {
        while (isRunning){
            sct.collect(number);
            number++;
            //每秒生成一条数据
        }
    }
}
```

```

        Thread.sleep(1000);
    }

}

@Override
public void cancel() {
    isRunning=false;
}
}

```

```

/**
 * 功能：从自定义的数据数据源里面获取数据，然后过滤出偶数
 */
public class StreamingDemowithMyNoPralalleSource {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<Long> numberStream = env.addSource(new
MyNoParalleSource()).setParallelism(1);
        SingleOutputStreamOperator<Long> dataStream = numberStream.map(new
MapFunction<Long, Long>() {
            @Override
            public Long map(Long value) throws Exception {
                System.out.println("接受到了数据: "+value);
                return value;
            }
        });
        SingleOutputStreamOperator<Long> filterDataStream =
dataStream.filter(new FilterFunction<Long>() {
            @Override
            public boolean filter(Long number) throws Exception {
                return number % 2 == 0;
            }
        });

        filterDataStream.print().setParallelism(1);
        env.execute("StreamingDemowithMyNoPralalleSource");
    }
}

```

运行结果:

接受到了数据: 1
接受到了数据: 2
2
接受到了数据: 3
接受到了数据: 4
4
接受到了数据: 5
接受到了数据: 6
6
接受到了数据: 7
接受到了数据: 8
8

4.1.4 自定义多并行度数据源

```
/**
 * 功能: 自定义支持并行度的数据源
 * 每秒产生一条数据
 */
public class MyParallelSource implements ParallelSourceFunction<Long> {
    private long number = 1L;
    private boolean isRunning = true;
    @Override
    public void run(SourceContext<Long> sct) throws Exception {
        while (isRunning){
            sct.collect(number);
            number++;
            //每秒生成一条数据
            Thread.sleep(1000);
        }
    }

    @Override
    public void cancel() {
        isRunning=false;
    }
}

public class StreamingDemowithMyPralalleSource {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<Long> numberStream = env.addSource(new
        MyParallelSource()).setParallelism(2);
        SingleOutputStreamOperator<Long> dataStream = numberStream.map(new
        MapFunction<Long, Long>() {
            @Override
            public Long map(Long value) throws Exception {
                System.out.println("接受到了数据: "+value);
                return value;
            }
        });
        SingleOutputStreamOperator<Long> filterDataStream =
        dataStream.filter(new FilterFunction<Long>() {
            @Override
```

```

        public boolean filter(Long number) throws Exception {
            return number % 2 == 0;
        }
    });

    filterDataStream.print().setParallelism(1);
    env.execute("StreamingDemoWithMyNoParallelSource");
}
}

```

运行结果:

```

接受到了数据: 1
接受到了数据: 1
接受到了数据: 2
接受到了数据: 2
2
2
接受到了数据: 3
接受到了数据: 3
接受到了数据: 4
4
接受到了数据: 4
4
接受到了数据: 5
接受到了数据: 5
接受到了数据: 6
接受到了数据: 6

```

4.2 常见Transformation操作

4.2.1 map和filter

```

/**
 * 数据源: 1 2 3 4 5.....源源不断过来
 * 通过map打印一下接受到数据
 * 通过filter过滤一下数据, 我们只需要偶数
 */
public class MapDemo {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<Long> numberStream = env.addSource(new
MyNoParallelSource()).setParallelism(1);
        SingleOutputStreamOperator<Long> dataStream = numberStream.map(new
MapFunction<Long, Long>() {
            @Override
            public Long map(Long value) throws Exception {
                System.out.println("接受到了数据: "+value);
                return value;
            }
        });
        SingleOutputStreamOperator<Long> filterDataStream =
dataStream.filter(new FilterFunction<Long>() {
            @Override
            public boolean filter(Long number) throws Exception {

```

```

        return number % 2 == 0;
    }
});

filterDataStream.print().setParallelism(1);
env.execute("StreamingDemowithMyNoParallelSource");
}

}

```

4.2.2 flatMap, keyBy和sum

```

/**
 * 滑动窗口实现单词计数
 * 数据源: socket
 * 需求: 每隔1秒计算最近2秒单词出现的次数
 *
 * 练习算子:
 * flatMap
 * keyBy:
 *   dataStream.keyBy("someKey") // 指定对象中的 "someKey"字段作为分组key
 *   dataStream.keyBy(0) //指定Tuple中的第一个元素作为分组key
 * sum
 */
public class windowWordCountJava {
    public static void main(String[] args) throws Exception {
        int port;
        try{
            ParameterTool parameterTool = ParameterTool.fromArgs(args);
            port = parameterTool.getInt("port");
        }catch (Exception e){
            System.err.println("no port set,user default port 9988");
            port=9988;
        }
        //步骤一: 获取flink运行环境 (stream)
        StreamExecutionEnvironment env=
StreamExecutionEnvironment.getExecutionEnvironment();
        String hostname="10.126.88.226";
        String delimiter="\n";
        //步骤二: 获取数据源
        DataStreamSource<String> textStream = env.socketTextStream(hostname,
port, delimiter);
        //步骤三: 执行transformation操作
        SingleOutputStreamOperator<WordCount> wordCountStream =
textStream.flatMap(new FlatMapFunction<String, WordCount>() {
            public void flatMap(String line, Collector<WordCount> out) throws
Exception {
                String[] fields = line.split("\t");
                for (String word : fields) {
                    out.collect(new WordCount(word, 1L));
                }
            }
        }).keyBy("word")
            .timeWindow(Time.seconds(2), Time.seconds(1))//每隔1秒计算最近2秒
            .sum("count");
    }
}

```

```

wordCountStream.print().setParallelism(1); //打印并设置并行度
//步骤四：运行程序
env.execute("socket word count");

}

public static class WordCount{
    public String word;
    public long count;
    public WordCount(){

    }
    public WordCount(String word, long count){
        this.word=word;
        this.count=count;
    }

    @Override
    public String toString() {
        return "WordCount{" +
            "word='" + word + '\'' +
            ", count=" + count +
            '}';
    }
}
}

```

4.2.3 union

```

/**
 * 合并多个流，新的流会包含所有流中的数据，但是union是一个限制，就是所有合并的流类型必须是一致的
 */
public class unionDemo {
    public static void main(String[] args) throws Exception {
        //获取Flink的运行环境
        StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();
        //获取数据源
        DataStreamSource<Long> text1 = env.addSource(new
        MyNoParallelSource()).setParallelism(1); //注意：针对此source，并行度只能设置为1
        DataStreamSource<Long> text2 = env.addSource(new
        MyNoParallelSource()).setParallelism(1);

        //把text1和text2组装到一起
        DataStream<Long> text = text1.union(text2);

        DataStream<Long> num = text.map(new MapFunction<Long, Long>() {
            @Override
            public Long map(Long value) throws Exception {
                System.out.println("原始接收到数据: " + value);
                return value;
            }
        });
        //每2秒钟处理一次数据
        DataStream<Long> sum = num.timeWindowAll(Time.seconds(2)).sum(0);
    }
}

```

```

        //打印结果
        sum.print().setParallelism(1);
        String jobName = unionDemo.class.getSimpleName();
        env.execute(jobName);
    }
}

```

4.2.4 connect,conMap和conFlatMap

```

/**
 * 和union类似，但是只能连接两个流，两个流的数据类型可以不同，会对两个流中的数据应用不同的处理方法
 */
public class ConnectionDemo {
    public static void main(String[] args) throws Exception {
        //获取Flink的运行环境
        StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();

        //获取数据源
        DataStreamSource<Long> text1 = env.addSource(new
        MyNoParallelSource()).setParallelism(1);//注意：针对此source，并行度只能设置为1

        DataStreamSource<Long> text2 = env.addSource(new
        MyNoParallelSource()).setParallelism(1);
        SingleOutputStreamOperator<String> text2_str = text2.map(new
        MapFunction<Long, String>() {
            @Override
            public String map(Long value) throws Exception {
                return "str_" + value;
            }
        });

        ConnectedStreams<Long, String> connectStream = text1.connect(text2_str);

        SingleOutputStreamOperator<Object> result = connectStream.map(new
        CoMapFunction<Long, String, Object>() {
            @Override
            public Object map1(Long value) throws Exception {
                return value;
            }

            @Override
            public Object map2(String value) throws Exception {
                return value;
            }
        });

        //打印结果
        result.print().setParallelism(1);
        String jobName = ConnectionDemo.class.getSimpleName();
        env.execute(jobName);
    }
}

```


4.3 常见sink操作

4.3.1 print() / printToErr()

打印每个元素的toString()方法的值到标准输出或者标准错误输出流中

4.3.2 writeAsText()

```
/**
 * 数据源: 1 2 3 4 5.....源源不断过来
 * 通过map打印一下接受到数据
 * 通过filter过滤一下数据, 我们只需要偶数
 */
public class WriteTextDemo {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<Long> numberStream = env.addSource(new
MyNoParallelSource()).setParallelism(1);
        SingleOutputStreamOperator<Long> dataStream = numberStream.map(new
MapFunction<Long, Long>() {
            @Override
            public Long map(Long value) throws Exception {
                System.out.println("接受到了数据: "+value);
                return value;
            }
        });
        SingleOutputStreamOperator<Long> filterDataStream =
dataStream.filter(new FilterFunction<Long>() {
            @Override
            public boolean filter(Long number) throws Exception {
                return number % 2 == 0;
            }
        });
        filterDataStream.writeAsText("D:\\nx\\flinklesson\\src\\output\\test").setParallelism(1);
        env.execute("StreamingDemoWithMyNoParallelSource");
    }
}
```

4.3.3 Flink提供的sink

- [Apache Kafka](#) (source/sink)
- [Apache Cassandra](#) (sink)
- [Amazon Kinesis Streams](#) (source/sink)
- [Elasticsearch](#) (sink)
- [Hadoop File System](#) (sink)
- [RabbitMQ](#) (source/sink)
- [Apache NiFi](#) (source/sink)
- [Twitter Streaming API](#) (source)
- [Google PubSub](#) (source/sink)

4.4 DataSet算子操作(了解即可)

4.4.1 source

基于文件

readTextFile(path)

基于集合

fromCollection(Collection)

4.4.2 transform

算子概览

Map: 输入一个元素, 然后返回一个元素, 中间可以做一些清洗转换等操作

FlatMap: 输入一个元素, 可以返回零个, 一个或者多个元素

MapPartition: 类似map, 一次处理一个分区的数据【如果在进行map处理的时候需要获取第三方资源链接, 建议使用MapPartition】

Filter: 过滤函数, 对传入的数据进行判断, 符合条件的数据会被留下

Reduce: 对数据进行聚合操作, 结合当前元素和上一次reduce返回的值进行聚合操作, 然后返回一个新的值

Aggregate: sum、max、min等

Distinct: 返回一个数据集中去重之后的元素, data.distinct()

Join: 内连接

OuterJoin: 外链接

Cross: 获取两个数据集的笛卡尔积

Union: 返回两个数据集的总和, 数据类型需要一致

First-n: 获取集合中的前N个元素

Sort Partition: 在本地对数据集的所有分区进行排序, 通过sortPartition()的链接调用来完成对多个字段的排序

MapPartition

```
public class MapPartitionDemo {
    public static void main(String[] args) throws Exception{

        //获取运行环境
        ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();
        ArrayList<String> data = new ArrayList<>();
        data.add("hello you");
        data.add("hello me");
        DataSource<String> text = env.fromCollection(data);

        /*text.map(new MapFunction<String, String>() {
            @Override
            public String map(String value) throws Exception {
                //获取数据库连接--注意, 此时是每过来一条数据就获取一次链接
                //处理数据
                //关闭连接
                return value;
            }
        });*/
        DataSet<String> mapPartitionData = text.mapPartition(new
        MapPartitionFunction<String, String>() {
            @Override
            public void mapPartition(Iterable<String> values, Collector<String>
            out) throws Exception {
                //获取数据库连接--注意, 此时是一个分区的数据获取一次连接【优点, 每个分区获取
                一次链接】
                //values中保存了一个分区的数据
```

```

        //处理数据
        Iterator<String> it = values.iterator();
        while (it.hasNext()) {
            String next = it.next();
            String[] split = next.split("\\w+");
            for (String word : split) {
                out.collect(word);
            }
        }
        //关闭链接
    }
});
mapPartitionData.print();
}
}

```

distinct

```

/**
 * 对数据进行去重
 */
public class DistinctDemo {
    public static void main(String[] args) throws Exception{

        //获取运行环境
        ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();
        ArrayList<String> data = new ArrayList<>();
        data.add("you jump");
        data.add("i jump");
        DataSource<String> text = env.fromCollection(data);
        FlatMapOperator<String, String> flatMapData = text.flatMap(new
        FlatMapFunction<String, String>() {
            @Override
            public void flatMap(String value, Collector<String> out) throws
            Exception {
                String[] split = value.toLowerCase().split("\\w+");
                for (String word : split) {
                    System.out.println("单词: "+word);
                    out.collect(word);
                }
            }
        });

        flatMapData.distinct()// 对数据进行整体去重
        .print();

    }
}

```

join

```

/**
 * 对数据进行join
 */

```

```

public class JoinDemo {
    public static void main(String[] args) throws Exception{

        //获取运行环境
        ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

        //tuple2<用户id, 用户姓名>
        ArrayList<Tuple2<Integer, String>> data1 = new ArrayList<>();
        data1.add(new Tuple2<>(1,"zs"));
        data1.add(new Tuple2<>(2,"ls"));
        data1.add(new Tuple2<>(3,"ww"));

        //tuple2<用户id, 用户所在城市>
        ArrayList<Tuple2<Integer, String>> data2 = new ArrayList<>();
        data2.add(new Tuple2<>(1,"beijing"));
        data2.add(new Tuple2<>(2,"shanghai"));
        data2.add(new Tuple2<>(3,"guangzhou"));

        DataSource<Tuple2<Integer, String>> text1 = env.fromCollection(data1);
        DataSource<Tuple2<Integer, String>> text2 = env.fromCollection(data2);

        text1.join(text2).where(0)//指定第一个数据集中需要比较的元素角标
            .equalTo(0)//指定第二个数据集中需要比较的元素角标
            .with(new JoinFunction<Tuple2<Integer,String>,
        Tuple2<Integer,String>, Tuple3<Integer,String,String>>() {
                @Override
                public Tuple3<Integer, String, String> join(Tuple2<Integer,
        String> first, Tuple2<Integer, String> second)
                    throws Exception {
                    return new Tuple3<>(first.f0,first.f1,second.f1);
                }
            }).print();

        System.out.println("=====");

        //注意，这里用map和上面使用的with最终效果是一致的。
        /*text1.join(text2).where(0)//指定第一个数据集中需要比较的元素角标
            .equalTo(0)//指定第二个数据集中需要比较的元素角标
            .map(new
        MapFunction<Tuple2<Tuple2<Integer,String>,Tuple2<Integer,String>>,
        Tuple3<Integer,String,String>>() {
                @Override
                public Tuple3<Integer, String, String>
        map(Tuple2<Tuple2<Integer, String>, Tuple2<Integer, String>> value) throws
        Exception {
                    return new Tuple3<>
        (value.f0.f0,value.f0.f1,value.f1.f1);
                }
            }).print();*/
    }
}

```

OutJoin

```
/**
 * 外连接：
 *      左外连接
 *      右外连接
 *      全外连接
 */
public class OuterJoinDemo {
    public static void main(String[] args) throws Exception{

        //获取运行环境
        ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();
        //tuple2<用户id, 用户姓名>
        ArrayList<Tuple2<Integer, String>> data1 = new ArrayList<>();
        data1.add(new Tuple2<>(1,"zs"));
        data1.add(new Tuple2<>(2,"ls"));
        data1.add(new Tuple2<>(3,"ww"));
        //tuple2<用户id, 用户所在城市>
        ArrayList<Tuple2<Integer, String>> data2 = new ArrayList<>();
        data2.add(new Tuple2<>(1,"beijing"));
        data2.add(new Tuple2<>(2,"shanghai"));
        data2.add(new Tuple2<>(4,"guangzhou"));
        DataSource<Tuple2<Integer, String>> text1 = env.fromCollection(data1);
        DataSource<Tuple2<Integer, String>> text2 = env.fromCollection(data2);
        /**
         * 左外连接
         *
         * 注意: second这个tuple中的元素可能为null
         *
         */
        text1.leftOuterJoin(text2)
            .where(0)
            .equalTo(0)
            .with(new JoinFunction<Tuple2<Integer,String>,
        Tuple2<Integer,String>, Tuple3<Integer,String,String>>() {
                @Override
                public Tuple3<Integer, String, String> join(Tuple2<Integer,
        String> first, Tuple2<Integer, String> second) throws Exception {
                    if(second==null){
                        return new Tuple3<>(first.f0,first.f1,"null");
                    }else{
                        return new Tuple3<>(first.f0,first.f1,second.f1);
                    }
                }
            }).print();
        System.out.println("=====");
        /**
         * 右外连接
         *
         * 注意: first这个tuple中的数据可能为null
         *
         */
        text1.rightOuterJoin(text2)
            .where(0)
```

```

        .equalTo(0)
        .with(new JoinFunction<Tuple2<Integer,String>,
Tuple2<Integer,String>, Tuple3<Integer,String,String>>() {
            @Override
            public Tuple3<Integer, String, String> join(Tuple2<Integer,
String> first, Tuple2<Integer, String> second) throws Exception {
                if(first==null){
                    return new Tuple3<>(second.f0,"null",second.f1);
                }
                return new Tuple3<>(first.f0,first.f1,second.f1);
            }
        }).print();

System.out.println("=====");

/**
 * 全外连接
 *
 * 注意: first和second这两个tuple都有可能为null
 *
 */

text1.fullOuterJoin(text2)
    .where(0)
    .equalTo(0)
    .with(new JoinFunction<Tuple2<Integer,String>,
Tuple2<Integer,String>, Tuple3<Integer,String,String>>() {
        @Override
        public Tuple3<Integer, String, String> join(Tuple2<Integer,
String> first, Tuple2<Integer, String> second) throws Exception {
            if(first==null){
                return new Tuple3<>(second.f0,"null",second.f1);
            }else if(second == null){
                return new Tuple3<>(first.f0,first.f1,"null");
            }else{
                return new Tuple3<>(first.f0,first.f1,second.f1);
            }
        }
    }).print();

    }
}

```

Cross

```

/**
 * 笛卡尔积
 */
public class CrossDemo {
    public static void main(String[] args) throws Exception{

        //获取运行环境
        ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();
    }
}

```

```

//tuple2<用户id, 用户姓名>
ArrayList<String> data1 = new ArrayList<>();
data1.add("zs");
data1.add("ww");
//tuple2<用户id, 用户所在城市>
ArrayList<Integer> data2 = new ArrayList<>();
data2.add(1);
data2.add(2);
DataSource<String> text1 = env.fromCollection(data1);
DataSource<Integer> text2 = env.fromCollection(data2);
CrossOperator.DefaultCross<String, Integer> cross = text1.cross(text2);
cross.print();

}
}

```

First-n 和 SortPartition

```

/**
 * TopN
 */
import java.util.ArrayList;

public class FirstNDemo {
    public static void main(String[] args) throws Exception{

        //获取运行环境
        ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

        ArrayList<Tuple2<Integer, String>> data = new ArrayList<>();
        data.add(new Tuple2<>(2, "zs"));
        data.add(new Tuple2<>(4, "ls"));
        data.add(new Tuple2<>(3, "ww"));
        data.add(new Tuple2<>(1, "xw"));
        data.add(new Tuple2<>(1, "aw"));
        data.add(new Tuple2<>(1, "mw"));

        DataSource<Tuple2<Integer, String>> text = env.fromCollection(data);

        //获取前3条数据，按照数据插入的顺序
        text.first(3).print();
        System.out.println("=====");

        //根据数据中的第一列进行分组，获取每组的前2个元素
        text.groupBy(0).first(2).print();
        System.out.println("=====");

        //根据数据中的第一列分组，再根据第二列进行组内排序[升序]，获取每组的前2个元素
        text.groupBy(0).sortGroup(1, Order.ASCENDING).first(2).print();
        System.out.println("=====");

        //不分组，全局排序获取集合中的前3个元素，针对第一个元素升序，第二个元素倒序
    }
}

```

```

text.sortPartition(0,Order.ASCENDING).sortPartition(1,Order.DECENDING).first(3)
).print();

    }
}

```

partition

```

/**
 * HashPartition
 *
 * RangePartition
 */
public class HashRangePartitionDemo {

    public static void main(String[] args) throws Exception{

        //获取运行环境
        ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

        ArrayList<Tuple2<Integer, String>> data = new ArrayList<>();
        data.add(new Tuple2<>(1,"hello1"));
        data.add(new Tuple2<>(2,"hello2"));
        data.add(new Tuple2<>(2,"hello3"));
        data.add(new Tuple2<>(3,"hello4"));
        data.add(new Tuple2<>(3,"hello5"));
        data.add(new Tuple2<>(3,"hello6"));
        data.add(new Tuple2<>(4,"hello7"));
        data.add(new Tuple2<>(4,"hello8"));
        data.add(new Tuple2<>(4,"hello9"));
        data.add(new Tuple2<>(4,"hello10"));
        data.add(new Tuple2<>(5,"hello11"));
        data.add(new Tuple2<>(5,"hello12"));
        data.add(new Tuple2<>(5,"hello13"));
        data.add(new Tuple2<>(5,"hello14"));
        data.add(new Tuple2<>(5,"hello15"));
        data.add(new Tuple2<>(6,"hello16"));
        data.add(new Tuple2<>(6,"hello17"));
        data.add(new Tuple2<>(6,"hello18"));
        data.add(new Tuple2<>(6,"hello19"));
        data.add(new Tuple2<>(6,"hello20"));
        data.add(new Tuple2<>(6,"hello21"));

        DataSource<Tuple2<Integer, String>> text = env.fromCollection(data);

        /*text.partitionByHash(0).mapPartition(new
        MapPartitionFunction<Tuple2<Integer,String>, Tuple2<Integer,String>>() {
            @Override
            public void mapPartition(Iterable<Tuple2<Integer, String>> values,
            Collector<Tuple2<Integer, String>> out) throws Exception {
                Iterator<Tuple2<Integer, String>> it = values.iterator();
                while (it.hasNext()){
                    Tuple2<Integer, String> next = it.next();

```



```

        System.out.println("当前线程
id: "+Thread.currentThread().getId()+" "+next);
    }
}
}).print();*/

    text.partitionByRange(0).mapPartition(new
MapPartitionFunction<Tuple2<Integer,String>, Tuple2<Integer,String>>() {
        @Override
        public void mapPartition(Iterable<Tuple2<Integer, String>> values,
Collector<Tuple2<Integer, String>> out) throws Exception {
            Iterator<Tuple2<Integer, String>> it = values.iterator();
            while (it.hasNext()){
                Tuple2<Integer, String> next = it.next();
                System.out.println("当前线程
id: "+Thread.currentThread().getId()+" "+next);
            }

        }
    }).print();

}
}
}

```

4.4.3 sink

writeAsText(): 将元素以字符串形式逐行写入，这些字符串通过调用每个元素的toString()方法来获取
writeAsCsv(): 将元组以逗号分隔写入文件中，行及字段之间的分隔是可配置的。每个字段的值来自对象的toString()方法
print(): 打印每个元素的toString()方法的值到标准输出或者标准错误输出流中

4.4.4 Flink之广播变量

广播变量允许编程人员在每台机器上保持1个只读的缓存变量，而不是传送变量的副本给tasks
广播变量创建后，它可以运行在集群中的任何function上，而不需要多次传递给集群节点。另外需要记住，不应该修改广播变量，这样才能确保每个节点获取到的值都是一致的
一句话解释，可以理解为一个公共的共享变量，我们可以把一个dataset 数据集广播出去，然后不同的task在节点上都能够获取到，这个数据在每个节点上只会存在一份。如果不使用broadcast，则在每个节点中的每个task中都需要拷贝一份dataset数据集，比较浪费内存(也就是一个节点中可能会存在多份dataset数据)。

用法

```

1: 初始化数据
DataSet<Integer> toBroadcast = env.fromElements(1, 2, 3)
2: 广播数据
withBroadcastSet(toBroadcast, "broadcastSetName");
3: 获取数据
Collection<Integer> broadcastSet =
getRuntimeContext().getBroadcastVariable("broadcastSetName");

```

注意:

- 1: 广播出去的变量存在于每个节点的内存中，所以这个数据集不能太大。因为广播出去的数据，会常驻内存，除非程序执行结束
- 2: 广播变量在初始化广播出去以后不支持修改，这样才能保证每个节点的数据都是一致的。

```

/**
 * broadcast广播变量
 * 需求：
 *   flink会从数据源中获取到用户的姓名
 *   最终需要把用户的姓名和年龄信息打印出来
 * 分析：
 *   所以就需要在中间的map处理的时候获取用户的年龄信息
 *   建议吧用户的关系数据集使用广播变量进行处理
 *
 */
public class BroadCastDemo {
    public static void main(String[] args) throws Exception{

        //获取运行环境
        ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();
        //1: 准备需要广播的数据
        ArrayList<Tuple2<String, Integer>> broadData = new ArrayList<>();
        broadData.add(new Tuple2<>("zs",18));
        broadData.add(new Tuple2<>("ls",20));
        broadData.add(new Tuple2<>("ww",17));
        DataSet<Tuple2<String, Integer>> tupleData =
        env.fromCollection(broadData);

        //1.1:处理需要广播的数据,把数据集转换成map类型, map中的key就是用户姓名, value就是
        用户年龄
        DataSet<HashMap<String, Integer>> toBroadcast = tupleData.map(new
        MapFunction<Tuple2<String, Integer>, HashMap<String, Integer>>() {
            @Override
            public HashMap<String, Integer> map(Tuple2<String, Integer> value)
            throws Exception {
                HashMap<String, Integer> res = new HashMap<>();
                res.put(value.f0, value.f1);
                return res;
            }
        });
        //源数据
        DataSource<String> data = env.fromElements("zs", "ls", "ww");
        //注意: 在这里需要使用到RichMapFunction获取广播变量
        DataSet<String> result = data.map(new RichMapFunction<String, String>()
        {
            List<HashMap<String, Integer>> broadCastMap = new
            ArrayList<HashMap<String, Integer>>();
            HashMap<String, Integer> allMap = new HashMap<String, Integer>();

            /**
             * 这个方法只会执行一次
             * 可以在这里实现一些初始化的功能
             * 所以, 就可以在open方法中获取广播变量数据
             */
            @Override
            public void open(Configuration parameters) throws Exception {
                super.open(parameters);
                //3: 获取广播数据
                this.broadCastMap =
                getRuntimeContext().getBroadcastVariable("broadCastMapName");
                for (HashMap map : broadCastMap) {

```

```

        allMap.putAll(map);
    }
}
@Override
public String map(String value) throws Exception {
    Integer age = allMap.get(value);
    return value + "," + age;
}
}).withBroadcastSet(toBroadcast, "broadcastMapName");//2: 执行广播数据的操作
result.print();
}
}

```

4.4.5 Flink之Counter (计数器)

Accumulator即累加器，与Mapreduce counter的应用场景差不多，都能很好地观察task在运行期间的数据变化

可以在Flink job任务中的算子函数中操作累加器，但是只能在任务执行结束之后才能获得累加器的最终结果。

Counter是一个具体的累加器(Accumulator)实现

IntCounter, LongCounter 和 DoubleCounter

用法

1: 创建累加器

```
private IntCounter numLines = new IntCounter();
```

2: 注册累加器

```
getRuntimeContext().addAccumulator("num-lines", this.numLines);
```

3: 使用累加器

```
this.numLines.add(1);
```

4: 获取累加器的结果

```
myJobExecutionResult.getAccumulatorResult("num-lines")
```

```

/**
 * 计数器
 */
public class CounterDemo {
    public static void main(String[] args) throws Exception{
        //获取运行环境
        ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();
        DataSource<String> data = env.fromElements("a", "b", "c", "d");
        DataSet<String> result = data.map(new RichMapFunction<String, String>()
        {

            //1:创建累加器
            private IntCounter numLines = new IntCounter();
            @Override
            public void open(Configuration parameters) throws Exception {
                super.open(parameters);
                //2:注册累加器
                getRuntimeContext().addAccumulator("num-lines",this.numLines);
            }
        }
    }
}

```

```

    }
    //int sum = 0;
    @Override
    public String map(String value) throws Exception {
        //如果并行度为1，使用普通的累加求和即可，但是设置多个并行度，则普通的累加求和
        //sum++;
        //System.out.println("sum: "+sum);
        this.numLines.add(1);
        return value;
    }
}).setParallelism(8);
//如果要获取counter的值，只能是任务
//result.print();
result.writeAsText("d:\\data\\mycounter");
JobExecutionResult jobResult = env.execute("counter");
//3: 获取累加器
int num = jobResult.getAccumulatorResult("num-lines");
System.out.println("num:"+num);
}
}

```

结果就不准了

六、总结 (5分钟)

1. 掌握常见的DataStream API
2. 掌握常见的DataSet API

七、作业

把所有的例子都敲一遍

八、互动